

Modeling with Java

lesson2

Once you have your environment configured and know how to develop an application with it, it's time to make your own project. In this lesson you will create all the entities required for your project in order to get your application working.

By now I assume that you know how to create a new entity with Eclipse, how to update the database schema any time you change your entities and how to run the application, because you have already read lesson 1, right?

2.1 Basic domain model

First, we'll create the entities for your Invoicing application. The domain model is rather basic, but enough to learn a lot of interesting things. Look at it in figure 2.1.

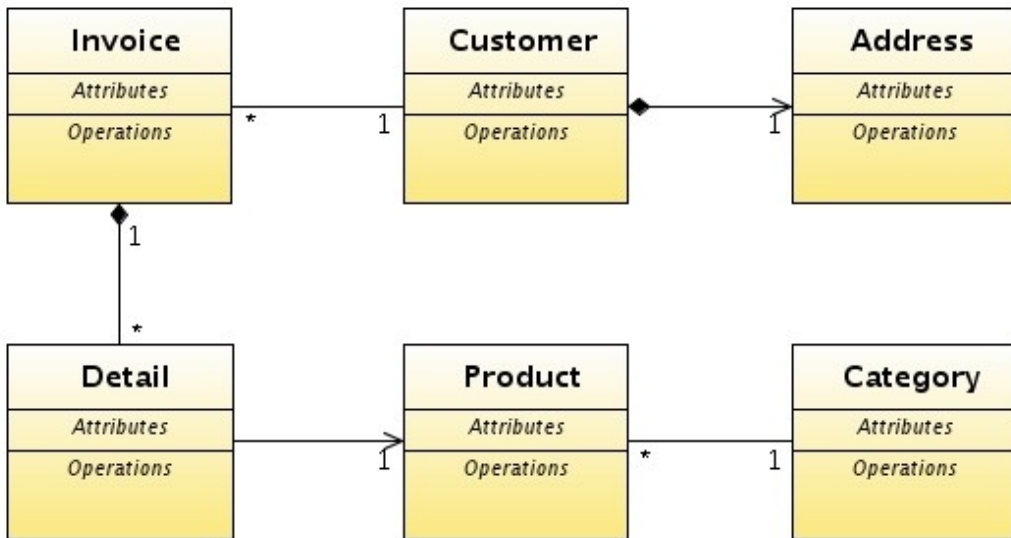


Figure 2.1 Initial UML diagram for Invoicing application

We'll start with six classes. Later on we'll add a few more to it. Remember that you already have an initial version of Customer and Product.

2.1.1 Reference (ManyToOne) as descriptions list (combo)

Let's start with the most simple case. We are going to create a Category entity and associate it to the Product, displaying it as a combo.

The code for Category entity is in listing 2.1.

21 Lesson 2: Modeling with Java

Listing 2.1 Category entity with UUID oid generation

```
package org.openxava.invoicing.model;

import javax.persistence.*;

import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity
public class Category {

    @Id
    @Hidden // The property is not shown to the user. It's an internal identifier
    @GeneratedValue(generator="system-uuid") // Universally Unique Identifier (1)
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;

    @Column(length=50)
    private String description;

    // Getters and setters
    ...
}
```

It only has an identifier and a description property. In this case we use the Universally Unique Identifier (shown as 1) algorithm to generate the id. The advantage of this id generator is that you can migrate your application to another database (DB2, MySQL, Oracle, Informix, etc) without touching your code. The other id generators of JPA rely on the database to generate the id thereby making them not so portable as UUID.

Execute the Category module and add some categories. Remember to update the database schema first.

Now, we'll associate Product with Category. Look at listing 2.2.

Listing 2.2 Product with a reference to Category

```
@Entity
public class Product {

    @Id @Column(length=9)
    private int number;

    @Column(length=50) @Required
    private String description;

    @ManyToOne( // The reference is persisted as a database relationship
        fetch=FetchType.LAZY, // The reference is loaded on demand
        optional=true) // The reference can have no value
    @DescriptionsList // Thus the reference is displayed using a combo
    private Category category; // A regular Java reference

    // Getters and setters
}
```

```
} ...
```

It's plain JPA's many-to-one relationship, the one that you can learn in appendix B. In this case, thanks to the `@DescriptionsList` annotation it is displayed as a combo (figure 2.2).

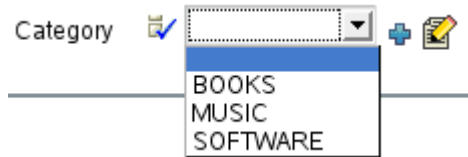


Figure 2.2 Category reference displayed as a combo

Now it's time to complete your Product entity.

2.1.2 Stereotypes

Product entity at least needs to have the attributes such as price. Also it would be nice to have photos and a field for remarks. We are going to use stereotypes to do it. A stereotype specifies a special use of a type. For example, you can use `String` to store names, comments or identifiers, and you can use a `BigDecimal` to store percentages, money or quantities. Different uses of stereotypes is to mark this specific use.

The best way to understand what a stereotype is, is to see it in action. Let's add price, photo, morePhotos and remarks properties to your Product entity (listing 2.3).

Listing 2.3 New properties for Product that use @Stereotype

```
@Stereotype("MONEY") // The price property is used to store money
private BigDecimal price; // BigDecimal is typically used for money

@Stereotype("PHOTO") // The user can view and change a photo
private byte [] photo;

@Stereotype("IMAGES_GALLERY") // A complete image gallery is available
@Column(length=32) // The 32 length string is for storing the key of the gallery
private String morePhotos;

@Stereotype("MEMO") // This is for a big text, a text area or equivalent will be used
private String remarks;

// Getters and setters
```

You have seen how to use stereotypes. Now you only have to use the name of the stereotype and OpenXava will apply special treatment. Execute the module for the Product now, and you will see the same as shown in figure 2.3.

23 Lesson 2: Modeling with Java

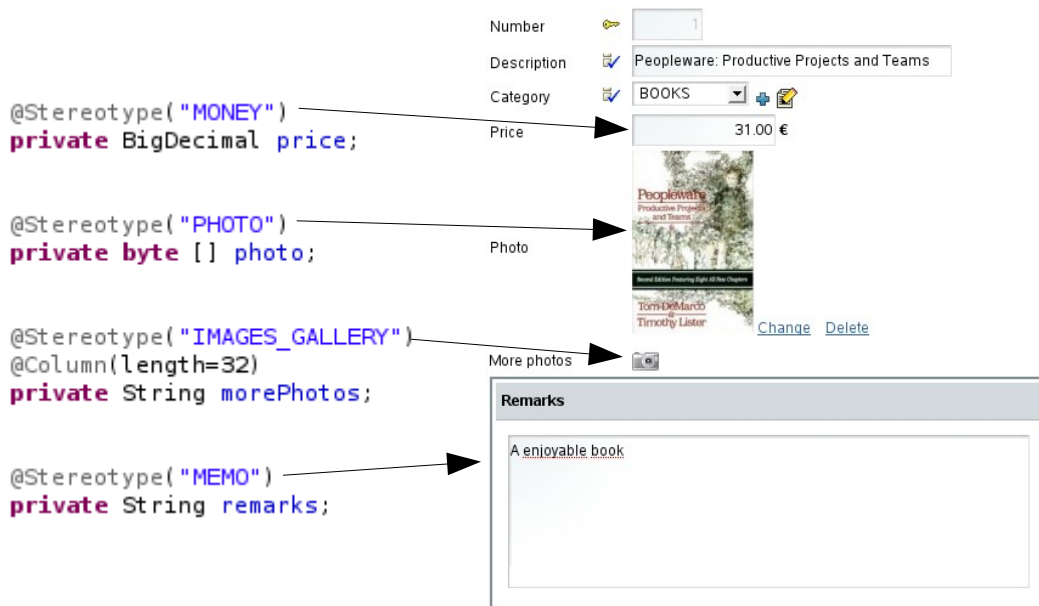


Figure 2.3 Visual effect of stereotypes in user interface

As you can see in figure 2.3, each stereotype produces an effect in the user interface. Stereotypes have effects in sizes, validations, editors, etc. Stereotypes also allow you to reuse built-in functionality easily. For example, with a mere marking of a simple string property as `@Stereotype("IMAGES_GALLERY")` you have a full image gallery available. Click on the camera of the `morePhotos` property and you'll see the gallery in figure 2.4.

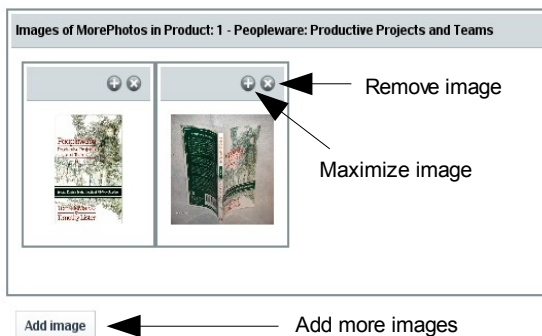


Figure 2.4 Images gallery produced by IMAGE_GALLERY stereotype

Apart from these, OpenXava has many other useful built in stereotypes such as `LABEL`, `BOLD_LABEL`, `DATETIME`, `ZERO_FILLED`, `HTML_TEXT`, `IMAGE_LABEL`, `EMAIL`, `TELEPHONE`, `WEBURL`, `IP`, `ISBN`, `CREDIT_CARD`, `EMAIL_LIST`.

Now you have Product ready to use. Let's refine the Customer now.

2.1.3 Embeddable

We are going to add address to our until now pretty naked Customer. The customer address is not shared by other customers, and when the customer is

removed his address is removed too. Therefore we'll model the address concept as an embeddable class in this case. You can learn this in section B.1.4 (from appendix B).

Add the Address class to your project. Its code is in listing 2.4.

Listing 2.4 Address is modeled as an embeddable class

```
@Embeddable // We use @Embeddable instead of @Entity
public class Address {

    @Column(length=30) // The members are annotated as in entity case
    private String street;

    @Column(length=5)
    private int zipCode;

    @Column(length=20)
    private String city;

    @Column(length=30)
    private String state;

    // Getters and setters
    ...
}
```

You can see how the regular class has been annotated as `@Embeddable`. Its properties are annotated in the same way as entities, though embeddable classes do not support all functionality of entities.

Now, you can use Address in any entity. Just add a reference to your Customer entity, leaving it as in listing 2.5.

Listing 2.5 Customer entity with a reference to the embeddable Address

```
@Entity
public class Customer {

    @Id
    @Column(length=6)
    private int number;

    @Column(length=50)
    @Required
    private String name;

    @Embedded // This is the way to reference an embeddable class
    private Address address; // A regular Java reference

    public Address getAddress() {
        if (address == null) address = new Address(); // Thus it never is null
        return address;
    }

    public void setAddress(Address address) {
```

25 Lesson 2: Modeling with Java

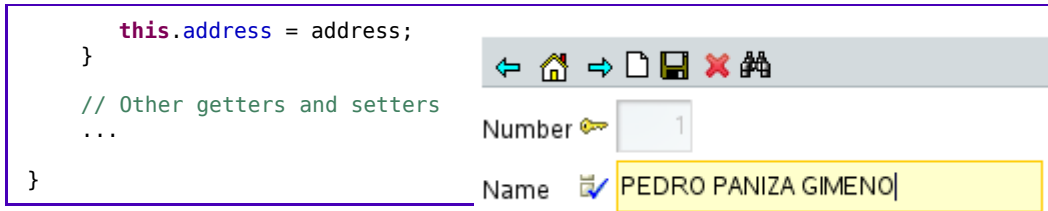


Figure 2.2 User interface for Customer entity with 2 properties

The Address data is stored in the same table as the Customer data. And from a user interface perspective you have a frame around address. If you do not like the frame you only have to annotate the reference with `@NoFrame` as shown in listing 2.6.

Listing 2.6 Address with `@NoFrame`

```
@Embedded @NoFrame // With @NoFrame no frame is shown for address
private Address address;
```

figure 2.5 shows the user interface for an embedded reference with and without `@NoFrame`.

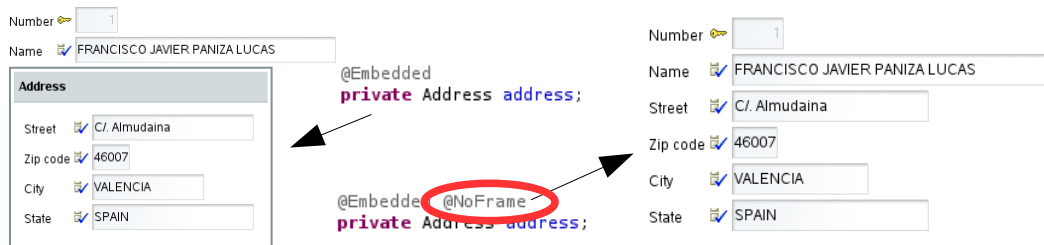


Figure 2.5 User interface for embedded reference with and without `@NoFrame`

Now that we have the basic entities running, it's time to face the core entity of your application, the Invoice entity. Let's start with it step by step.

2.1.4 Composite key

We are not going to use a composite key for Invoice. It's better to avoid the use of composite keys. You always have the option to use an auto generated hidden id. Although, sometimes you may need to connect to a legacy database or maybe the schema design has been done by someone that likes composite keys, and you must use composite keys albeit it's not the best choice. Therefore, here you will learn how to use a composite key, though at the end we'll change it to a single auto generated id.

Let's start with a simple version of Invoice entity. See it in listing 2.7.

Listing 2.7 First version of Invoice, using a composite key

```

@Entity
@IdClass(InvoiceKey.class) // The id class has all the key properties (1)
public class Invoice {

    @Id // Though we have id class it still needs to be marked as @Id (2)
    @Column(length=4)
    private int year;

    @Id // Though we have id class it still needs to be marked as @Id (2)
    @Column(length=6)
    private int number;

    @Required
    private Date date;

    @Stereotype("MEMO")
    private String remarks;

    // Getters and setters

}

```

If you want to use year and number as a composite key for your Invoice, a way to do so is by marking year and number as @Id (shown as 2), and also to have an id class (shown as 1). The id class must have year and number as properties. You can see it in listing 2.8.

Listing 2.8 InvoiceKey: The id class for Invoice

```

public class InvoiceKey
    implements java.io.Serializable { // The key class must be serializable

    private int year; // It contains the properties marked ...
    private int number; // ... as @Id in the entity

    @Override
    public boolean equals(Object obj) { // It must define equals method
        if (obj == null) return false;
        return obj.toString().equals(this.toString());
    }

    @Override
    public int hashCode() { // It must define hashCode method
        return toString().hashCode();
    }

    @Override
    public String toString() {
        return "InvoiceKey::" + year + ":" + number;
    }

    // Getters and setters for year and number

}

```

The listing 2.8 shows some of the requirements of a primary key class, such as to be serializable and to implement hashCode() and equals().

27 Lesson 2: Modeling with Java

You already know how to use a composite key, and given that we have control over our schema, in the end we are going to use a UUID identifier for Invoice. Rewrite the Invoice entity and leave it as shown in listing 2.9.

Listing 2.9 Invoice using a UUID single key

```
package org.openxava.invoicing.model;

import java.util.*;
import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity // @IdClass removed
public class Invoice {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid; // Added as hidden autogenerated key.
                       // Remember to add getOid() and setOid()

    @Column(length=4) // @Id removed
    @DefaultValueCalculator(CurrentYearCalculator.class)
    private int year;

    @Column(length=6) // @Id removed
    @DefaultValueCalculator(value=NextNumberForYearCalculator.class,
        properties=@PropertyValue(name="year")
    )
    private int number;

    ...
}
```

Also delete the InvoiceKey class. Using a hidden auto-generated key for Invoice class has several practical advantages over a composite key: You do not have to write the boring InvoiceKey class, you can modify the invoice number without losing any association from other objects and you can store in the same table the orders and invoices with the year/number repeated.

The code you have is enough to run the Invoice module. Do it and add some invoices if you want. Still a lot of work is remaining to be done in Invoice, like the default values for year, number and date.

2.1.5 Calculating default values

Currently the user needs to type the year, number and date in order to enter an invoice. It would be nice to have default values for them. It's easy to do it using the @DefaultValueCalculator annotation. In listing 2.10 you see how we can add the default values for year and date:

Listing 2.10 Using OpenXava built-in default value calculators for year and date

```

@Column(length=4)
@DefaultValueCalculator(CurrentYearCalculator.class) // Current year
private int year;

@Required
@DefaultValueCalculator(CurrentDateCalculator.class) // Current date
private Date date;

```

From now on when the user clicks on the 'new' button the year field will have the current year, and the date field the current date. These two calculators (CurrentYearCalculator and CurrentDateCalculator) are included in OpenXava. You can explore the `org.openxava.calculators` package to see other useful built-in calculators.

But sometimes you need your own logic for calculating the default value. For example, for number we want to add one to the last invoice number in the same year. Creating your own calculator with your logic is easy. First, create a package for calculators and call it `org.openxava.invoicing.calculators`. Then create in it a `NextNumberForYearCalculator` class, with the code from listing 2.11.

Listing 2.11 Custom calculator for default value of invoice number

```

package org.openxava.invoicing.calculators;

import javax.persistence.*;
import org.openxava.calculators.*;
import org.openxava.jpa.*;

public class NextNumberForYearCalculator
    implements ICalculator { // A calculator must implement ICalculator

    private int year; // This value will be injected (using its setter) before calculating

    public Object calculate() throws Exception { // It does the calculation
        Query query = XPersistence.getManager() // A JPA query
            .createQuery("select max(i.number) from Invoice i" +
                " where i.year = :year"); // The query returns the max
                                                // invoice number of the indicated year
        query.setParameter("year", year); // We use the injected year as a
                                                // parameter for the query
        Integer lastNumber = (Integer) query.getSingleResult();
        return lastNumber == null?1:lastNumber + 1; // Returns the last invoice number
                                                // of the year + 1 or 1 if there is no last number
    }

    public int getYear() {
        return year;
    }

    public void setYear(int year) {
        this.year = year;
    }
}

```

```
}

```

Your calculator must implement `ICalculator` interface (and therefore have a `calculate()` method). We declare a year property to put in the year of the calculation. To implement the logic we use a JPA query. You can learn how to use JPA in appendix B. Now we only have to annotate the number property in the Invoice entity (listing 2.12).

Listing 2.12 Invoice number property annotated with a custom calculator

```
@Id @Column(length=6)
@DefaultValueCalculator(value=NextNumberForYearCalculator.class,
    properties=@PropertyValue(name="year") // To inject the value of year from Invoice to
                                           // the calculator before calling to calculate()
)
private int number;
```

In this case you see something new, an annotation `@PropertyValue`. By using this annotation you're saying that the value of year property of the current Invoice will be moved to the property year of the calculator before doing the calculation. Now when ever the user clicks on 'new' the next invoice number is available for the year field. The way of calculating the invoice number is not the best for many concurrent users adding invoices. Don't worry, we'll improve this issue later on.

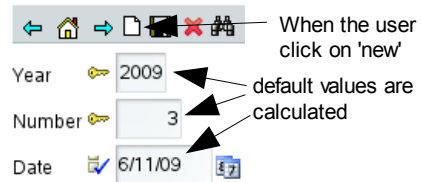


Figure 2.6 Effect of default value calculators

You can see the visual effect of the default value calculators in figure 2.6. Default values are only the initial values. The user can change them if he wishes to.

2.1.6 Regular reference (*ManyToOne*)

Now that we have all atomic properties ready to use it's time to add relationships with other entities. We'll begin adding a reference from Invoice to Customer, because an invoice without customer is not very useful. Add the code in listing 2.13 to the Invoice entity.

Listing 2.13 Reference to Customer from Invoice

```
@ManyToOne(fetch=FetchType.LAZY, optional=false) // Customer is required
private Customer customer;

// Getters and setters for customer
```

Nothing more is required. However, before trying the new Invoice module

you have to remove the current invoices (executing 'DELETE FROM Invoice'), and then updating the schema. Deleting the previous added invoices is necessary because the optional=false in the @ManyToOne annotation does not allow invoices with no customer².

The Invoice module is like the one in figure 2.7.

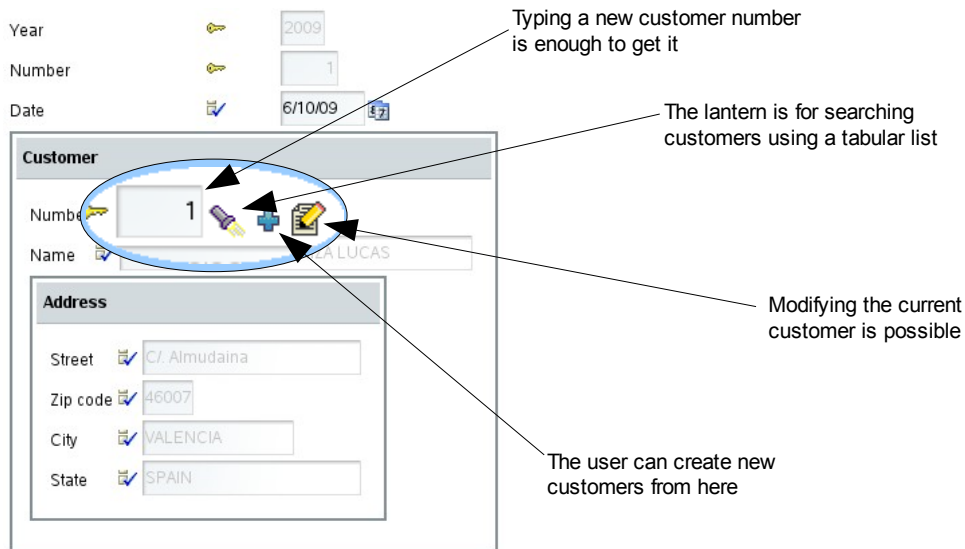


Figure 2.7 User interface for customer reference from Invoice

There is no more work left here now. Let's add the collection of details to your Invoice.

2.1.7 Collection of dependent entities (ManyToOne with cascade)

Usually an invoice needs to have a couple of lines with the details of products, quantities, etc. These details are part of the invoice. They are not shared with other invoices and when an invoice is deleted its details are also deleted. So, the more natural way of modeling the invoice details is to use embeddable object. Unfortunately, JPA (at least by 1.0) does not support collection of embeddable objects. Fortunately this is not a real problem, because you can use a collection of entities with cascade type REMOVE or ALL. Let's look at it for the case of the details collection definition in the Invoice entity as shown in listing 2.14.

Listing 2.14 Collection of details in Invoice

```
@OneToMany( // To declare this as a persistent collection
    mappedBy="parent", // The member of Detail that stores the relationship
```

² If you want to preserve the current data put optional=true, update schema, update the customer data for the existing invoices and put optional=false again

31 Lesson 2: Modeling with Java

```
        cascade=CascadeType.ALL) // Indicates this is a collection of dependent entities
    private Collection<Detail> details = new ArrayList<Detail>();

    // Getter and setter for details
```

Using `cascade=CascadeType.ALL` when the invoice is removed its details are removed too. Or if you add a new detail to an already persistent invoice, the detail is saved automatically, and when you mark an invoice as persistent its details are also marked as persistent, and so on. We can say that the entities of details collection are dependent on Invoice. This is good enough to simulate embeddable semantic using entities. You can also use `cascade=CascadeType.REMOVE` to define a dependent collection.

In order to make this collection work you need to write the Detail class (listing 2.15).

Listing 2.15 First version of Detail line entity for Invoice

```
package org.openxava.invoicing.model;

import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity
public class Detail {

    @ManyToOne // Without lazy fetching because it fails when removing a detail from parent
    private Invoice parent; // Thus the relationship between Detail and
                          // Invoice is bidirectional

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid; // UUID identifier, see Category entity (Listing 2.1)

    private int quantity;

    @ManyToOne(fetch=FetchType.LAZY, optional=true)
    private Product product;

    // Getters and setters
    ...
}
```

At the moment we only have quantity and product and that is enough to get the Invoice running with details. You can see in figure 2.8 how the user can add, edit and remove elements from the collection, filter and order the data and export to PDF and Excel.

But, figure 2.8 emphasizes that the properties to show by default in a collection are the plain ones, that is the properties of references are not included by default. This fact produces an ugly user interface for our collection of invoice details, because only the quantity property is shown.

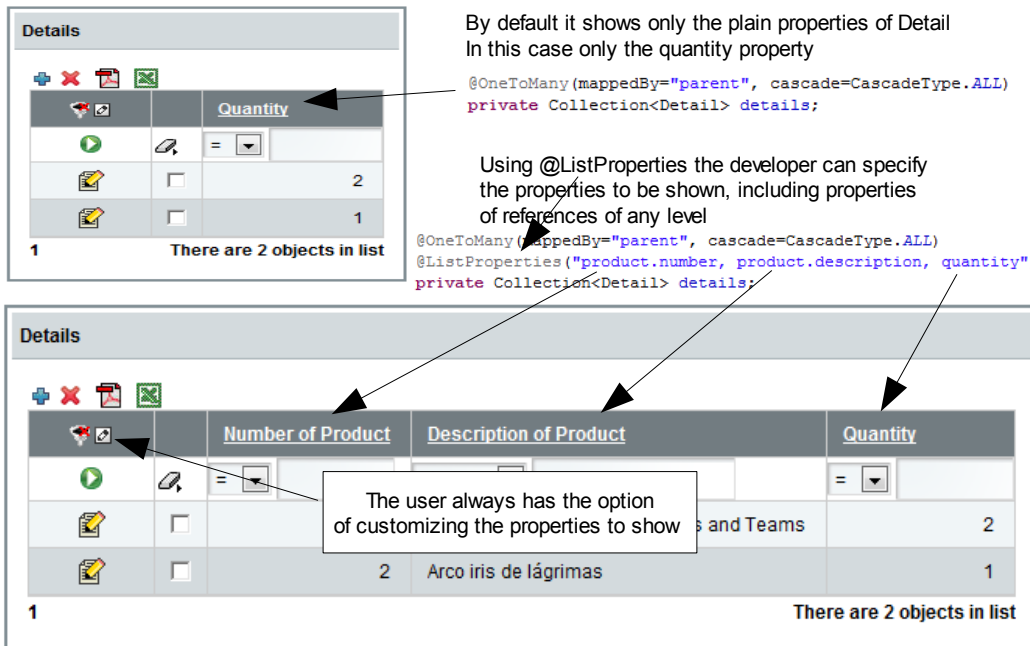


Figure 2.8 Effect of @ListProperties in the user interface of details collection

Figure 2.8 shows how to use the @ListProperties annotation to define the properties to be shown in the user interface of a collection initially. We say “initially” because the user can customize the properties of the collection to be shown in order to adapt the collection data to his needs or preferences. @ListProperties is easy to use, see it in listing 2.16.

Listing 2.16 @ListProperties for defining the properties to show in the collection

```
@OneToMany(mappedBy="parent", cascade=CascadeType.ALL)
@ListProperties("product.number, product.description, quantity")
private Collection<Detail> details = new ArrayList<Detail>();
```

As you can see, you only have to feed the value for the annotation @ListProperties with the list of the properties you wish, separated by commas. You can use qualified properties, that is, to use the dot notation for accessing properties of references, such as product.number and product.description in this case.

2.2 Refining the user interface

Congratulations! You have finished your domain model classes, and you have an application running. Now the user can work with products, categories, customers and even create invoices. In the case of products, categories and customers the user interface is pretty decent, though the user interface for

33 Lesson 2: Modeling with Java

invoices still can be improved a little.

By the way, you already have used some OpenXava annotations for refining the presentation, such as `@DescriptionsList`, `@NoFrame` and `@ListProperties`. In this section we'll use more of these annotations to give the Invoice user interface a better look without too much effort.

2.2.1 Default user interface

Figure 2.9 shows the default user interface for Invoice. As you see, OpenXava shows all the members, one per row, in the order you have declared them in the source code. Also, you can see how in the case of the customer reference the default view of the Customer is created.

```
public class Invoice {  
  
    private int year;..  
  
    private int number;..  
  
    private Date date;..  
  
    private Customer customer;..  
  
    private Collection<Detail> details;..  
  
    private String remarks;..  
}
```

By default, all the members of Invoice are shown in UI just in the order they have been defined in source code

Customer reference uses its default view, which is too much extensive

Number of Product	Description of Product	Quantity
1	Peopleware: Productive Projects and Teams	2
2	Arco Iris de lágrimas	1

There are 2 objects in list

Figure 2.9 Default user interface generated automatically for Invoice

We are going to do some improvements. First, we'll define the layout of the members explicitly. In this way we can put year, number and date in the same row. Second, we are going to use a simpler view for customer. The user does not need to see all the data of the customer when he is entering the invoice.

2.2.2 Using `@View` for defining layout

For defining the layout of Invoice members in the user interface you have to

use the `@View` annotation. It is easy because you only have to enumerate the members to be shown. Look at the code in listing 2.17.

Listing 2.17 @View annotation for defining the layout of Invoice members

```
@View(members= // This view has no name, so it will be the view used by default
    "year, number, date;" + // Comma separated means in the same line
    "customer;" + // Semicolon means a new line
    "details;" +
    "remarks"
)
public class Invoice {
```

At the end, we show all the members of Invoice, but we use commas to separate year, number and date. Thus they are in the same line, producing a more compact user interface, as you can see in figure 2.10.



Figure 2.10 Several members in the same line

2.2.3 Using @ReferenceView to refine the user interface for reference

You still need to refine the way the customer reference is displayed, because it displays all the members of Customer, and for entering data for an Invoice, a simpler view of the customer may be better. To do so, you have to define a Simple view in Customer, and then indicate in Invoice that you want to use the Simple view of Customer to display it.

First, let's define the Simple view in Customer, just as in listing 2.18.

Listing 2.18 A Simple view for Customer, with only number and name

```
@View(name="Simple", // This view is used only when "Simple" is specified
    members="number, name" // Shows only number and name in the same line
)
public class Customer {
```

When a view has a name, as in this case, then that view is only used when that name is specified. That is, though Customer has only this `@View` annotation, when you try to display a Customer it will not use this Simple view, but the one generated by default. If you define a `@View` with no name, that view will be the default one, though this is not the case.

Now, you have to indicate that the reference to Customer from Invoice must use this Simple view. This is done by means of `@ReferenceView`. See the listing 2.19.

Listing 2.19 Using @ReferenceView to specify the view to use for the reference

```
@ManyToOne(fetch=FetchType.LAZY, optional=false)
@ReferenceView("Simple") // The view named 'Simple' is used to display this reference
private Customer customer;
```

Really simple, you only have to indicate the name of the view of the referenced entity you want to use.

After this the customer reference will be shown in a more compact way, just as you can see in figure 2.11.

figure 2.11 Reference to Customer using its Simple view

You can see that you have refined your Invoice interface a little bit.

2.2.4 Refining a collection item entry

You already have done some refinement of collection user interface using the @ListProperties annotation. Still, the look of the collection when the user adds a new item is somewhat clumsy. Look at figure 2.12.

The default user interface to enter a new Detail is clumsy

It shows all data from product, moreover inside a frame

Maybe the photo is not necessary now

The remarks can be omitted too

We want a simpler interface to enter a new invoice line. With only the minimum data, in the same line, without inner frames and with appropriate labels

Figure 2.12 Default invoice detail entry is clumsy, a simpler one would be better

The way to simplify detail invoice entry is by using `@View` and `@ReferenceView`.

Let's start defining the default view for the `Detail` entity. Just as in listing 2.20.

Listing 2.20 Defining a default view for Detail

```
@Entity
@View(members="product, quantity") // In the same line, because it's comma separated
public class Detail {
```

As you can see, we change the order, first product then quantity, and we put both in the same line. Still product is displayed using too big a view. You can fix it using `@ReferenceView`, as you see it in listing 2.21.

Listing 2.21 Reference to Product in Detail with `@ReferenceView` and `@NoFrame`

```
@ManyToOne(fetch=FetchType.LAZY, optional=true)
@ReferenceView("Simple") // Product is displayed using its Simple view
@NoFrame // No frame is used around product data
private Product product;
```

Note that we also use `@NoFrame` in order to eliminate the frame around product, then integrate visually the quantity with the product number and the description.

Of course, you need a view called `Simple` in `Product` entity. Let's see it in listing 2.22.

Listing 2.22 View Simple in Product to be used from Detail

```
@Entity
@View(name="Simple",
      members="number, description") // number and description in the same line
public class Product {
```

Now, the detail will contain the product number, product description and quantity in the same line. A neat interface.

Because we use no frame for product, the product properties are seen by the user as properties of `Detail`. It would be useful to change the default labels for number and description of product for this view `Simple` to be clear to the user. Just as shown in figure 2.13.

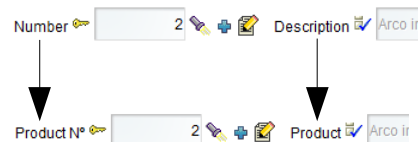


Figure 2.13 Better labels for product properties needed

It is easy to set the labels for properties of an entity in a concrete view. You only need to edit the `Invoicing-labels_en.properties` file located at `/Invoicing/i18n`, and add the lines of listing 2.23.

Listing 2.23 Entries in Invoice-labels_en.properties for label definition

```
Product.views.Simple.number=Product Nº
Product.views.Simple.description=Product
```

As you can see, to define the label you use dot notation for the key, defining first the entity name (Product in this case), then “views”, then the name of the view and the name of the the property. Of course, as value put your desired label.

2.2.5 Refined user interface

figure 2.14 shows the result of our refinements in the Invoice user interface.

The screenshot shows the Invoice user interface with the following components and annotations:

- Customer Section:** Includes fields for Year (2009), Number (1), and Date (6/25/09). Below these is a form for Customer details with fields for Number (1) and Name (PEDRO PANIZA GIMENO).
- Details Section:** A table with columns 'Number of Product' and 'Description'. It contains two rows:

Number of Product	Description
1	Peopleware: Productive Projects and Teams
2	Arco Iris de lágrimas

 An annotation points to the table with the text: "The 'Simple' view is used to display the customer reference in Invoice".
- Remarks Section:** A text area for entering remarks.
- Annotations:**
 - "Comma separated means in the same line" points to the @View annotation in the code.
 - "'Simple' is a view of Customer" points to the @ReferenceView annotation in the code.

```
@View(members="year, number, date;" +
      "customer;" +
      "details;" +
      "remarks"
    )
public class Invoice {
    @ReferenceView("Simple")
    private Customer customer;
}

@View(name="Simple",
      members="number, name"
    )
public class Customer {
```

Figure 2.14 User interface for Invoice refined with @View and @ReferenceView

You have seen how easy it is to use @View and @ReferenceView to get a more compact user interface for Invoice.

Now you have a user interface good enough to start working, and you really have done little work to get it.

2.3 Agile development

Nowadays agile development is no longer a “new and breaking technique”, but an established way to do software development, even the ideal way to go for many people.

If you are not familiar with agile development you can have a look at www.agilemanifesto.org. Basically, agile development encourages the use of feedback from a working product over a careful upfront design. This gives a more prominent role to programmers and users, and minimizes the importance of

analysts and software architects.

This type of development also needs a different type of tools. Because you need a working application rapidly. It must be as rapid to develop the initial application as it would be writing the functional description. Moreover, you need to respond to the user feedback quickly. The user needs to see his proposals running in short time.

OpenXava is ideal for agile development because not only does it allow a very rapid initial development, but it also allows you to make changes and see the effects instantly. Let's see a little example of this.

For example, once the user has looked at your application and starts to play with it, he takes into account that he works with books, music, software and so on. All these products have an author, and it would be useful to store the author, and see products by author.

Adding this new feature to your application is simple and rapid. First, create a new class for Author, with the code from listing 2.24.

Listing 2.24 Code for the Author entity

```
package org.openxava.invoicing.model;

import javax.persistence.*;
import org.hibernate.annotations.GenericGenerator;
import org.openxava.annotations.*;

@Entity
public class Author {

    @Id @GeneratedValue(generator="system-uuid") @Hidden
    @GenericGenerator(name="system-uuid", strategy = "uuid")
    @Column(length=32)
    private String oid;

    @Column(length=50) @Required
    private String name;

    // Getters and setters
    ...

}
```

Now, add the code of listing 2.25 to the existing Product entity.

Listing 2.25 Reference to Author from Product

```
@ManyToOne(fetch=FetchType.LAZY)
@DescriptionsList
private Author author;

public void setAuthor(Author author) {
    this.author = author;
}
```

```
public Author getAuthor() {
    return author;
}
```

Thus, your Product entity has a reference to Author.

Really you have written a little amount of code. In order to see the effect, you only need to build your project (just press Ctrl-B in your Eclipse), which is just immediate; update the database schema, executing the updateSchema ant target, just a few seconds. Go to the browser and reload the page with the Product module, and you will see there, a combo for choosing the author of the product, just as you see in figure 2.15.

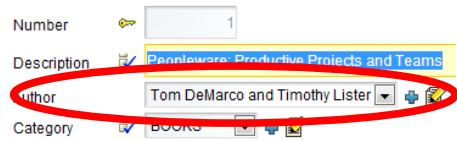


Figure 2.15 Reference to Author from Product

What if the user wants to choose an author and see all his products? Well. This is plain vanilla. You only have to make the relationship between Product and Author bidirectional. Go to the Author class and add the code in listing 2.26.


Listing 2.26 Collection of products in Author entity

```
@OneToMany(mappedBy="author")
@ListProperties("number, description, price")
private Collection<Product> products;





public Collection<Product> getProducts() {
    return products;
}








public void setProducts(Collection<Product> products) {
    this.products = products;
}
```

Now, press Ctrl+B (to build) and refresh the browser with Author module. Choose an author and you will see his products. You have to see something like the figure 2.16. Yes, you add a new collection, refresh your browser and there you get the full user interface to manage it.

Name  Tom DeMarco and Timothy Lister

Products

		Number	Description	Price
		= 	starts 	= 
	<input type="checkbox"/>	1	Peopleware: Productive Projects and Teams	31.00
	<input type="checkbox"/>	3	Adrenaline Junkies and Template Zombies	35.00

1

There are 2 objects in list

Figure 2.16 User interface for Author with a products collection

In this case it was not necessary to update the schema. You only have to do it when the database structure needs to be changed, and this usually occurs when you add, remove or rename fields or relationships from entities, or create new entities. The case of products collection is an exception, because the column needed for the relationship is already in place in the table for products. Anyways, don't bother much about this, if in doubt, just update schema, there's no harm.

In this section you have the complete code and steps required to do changes and see the result in the most interactive way. You have seen how OpenXava is an agile tool, ideal for doing agile development.

2.4 Summary

In this lesson you have learned how to use simple Java classes to create a Java Web application. With a mere few Java classes required to define your domain, you have a running application. Also, you have learned how to refine the default user interface using some OpenXava annotations.

Yes! Now you have a working application with little effort. Although this application “as is” can be useful as a CRUD utility or a prototype, you still need to add validations, business logic, user interface behavior, security and so on in order to convert these entities you have written into a business application ready for your user.

You will learn all these advanced topics in the forthcoming lessons.